# trace manual page - Tcl Built-In Commands

tcl-lang.org/man/tcl/TclCmd/trace.htm

## NAME

trace — Monitor variable accesses, command usages and command executions

## SYNOPSIS

**trace** *option* ?*arg arg ...*?

## DESCRIPTION

This command causes Tcl commands to be executed whenever certain operations are invoked. The legal *option*s (which may be abbreviated) are:

**trace add *type name ops ?args?***

Where *type* is **command**, **execution**, or **variable**.

**trace add command *name ops commandPrefix***

Arrange for *commandPrefix* to be executed (with additional arguments) whenever command *name* is modified in one of the ways given by the list *ops*. *Name* will be resolved using the usual namespace resolution rules used by commands. If the command does not exist, an error will be thrown.

*Ops* indicates which operations are of interest, and is a list of one or more of the following items:

**rename**

Invoke *commandPrefix* whenever the traced command is renamed. Note that renaming to the empty string is considered deletion, and will not be traced with “**rename**”.

**delete**

Invoke *commandPrefix* when the traced command is deleted. Commands can be deleted explicitly by using the **rename** command to rename the command to an empty string. Commands are also deleted when the interpreter is deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, depending on the operations being traced, a number of arguments are appended to *commandPrefix* so that the actual command is as follows:

```
commandPrefix oldName newName op
```

*OldName* and *newName* give the traced command's current (old) name, and the name to which it is being renamed (the empty string if this is a “delete” operation). *Op* indicates what operation is being performed on the command, and is one of **rename** or **delete** as defined above. The trace operation cannot be used to stop a command from being deleted. Tcl will always remove the command once the trace is complete. Recursive renaming or deleting will not cause further traces of the same type to be evaluated, so a

delete trace which itself deletes the command, or a rename trace which itself renames the command will not cause further trace evaluations to occur. Both *oldName* and *newName* are fully qualified with any namespace(s) in which they appear.

**trace add execution *name ops commandPrefix***
Arrange for *commandPrefix* to be executed (with additional arguments) whenever command *name* is executed, with traces occurring at the points indicated by the list *ops*. *Name* will be resolved using the usual namespace resolution rules used by commands. If the command does not exist, an error will be thrown.
*Ops* indicates which operations are of interest, and is a list of one or more of the following items:

**enter**
Invoke *commandPrefix* whenever the command *name* is executed, just before the actual execution takes place.

**leave**
Invoke *commandPrefix* whenever the command *name* is executed, just after the actual execution takes place.

**enterstep**
Invoke *commandPrefix* for every Tcl command which is executed from the start of the execution of the procedure *name* until that procedure finishes. *CommandPrefix* is invoked just before the actual execution of the Tcl command being reported takes place. For example if we have “proc foo {} { puts "hello" }”, then an *enterstep* trace would be invoked just before “*puts "hello"*” is executed. Setting an *enterstep* trace on a command *name* that does not refer to a procedure will not result in an error and is simply ignored.

**leavestep**
Invoke *commandPrefix* for every Tcl command which is executed from the start of the execution of the procedure *name* until that procedure finishes. *CommandPrefix* is invoked just after the actual execution of the Tcl command being reported takes place. Setting a *leavestep* trace on a command *name* that does not refer to a procedure will not result in an error and is simply ignored.

When the trace triggers, depending on the operations being traced, a number of arguments are appended to *commandPrefix* so that the actual command is as follows:

For **enter** and **enterstep** operations:

```
commandPrefix command-string op
```

*Command-string* gives the complete current command being executed (the traced command for a **enter** operation, an arbitrary command for a **enterstep** operation), including all arguments in their fully expanded form. *Op* indicates what operation is being performed on the command execution, and is one of **enter** or **enterstep** as defined

above. The trace operation can be used to stop the command from executing, by deleting the command in question. Of course when the command is subsequently executed, an "invalid command" error will occur.

For **leave** and **leavestep** operations:

```
commandPrefix command-string code result op
```

*Command-string* gives the complete current command being executed (the traced command for a **enter** operation, an arbitrary command for a **enterstep** operation), including all arguments in their fully expanded form. *Code* gives the result code of that execution, and *result* the result string. *Op* indicates what operation is being performed on the command execution, and is one of **leave** or **leavestep** as defined above. Note that the creation of many **enterstep** or **leavestep** traces can lead to unintuitive results, since the invoked commands from one trace can themselves lead to further command invocations for other traces.

*CommandPrefix* executes in the same context as the code that invoked the traced operation: thus the *commandPrefix*, if invoked from a procedure, will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *commandPrefix* invokes a procedure (which it normally does) then the procedure will have to use **upvar** or **uplevel** commands if it wishes to access the local variables of the code which invoked the trace operation.

While *commandPrefix* is executing during an execution trace, traces on *name* are temporarily disabled. This allows the *commandPrefix* to execute *name* in its body without invoking any other traces again. If an error occurs while executing the *commandPrefix*, then the command *name* as a whole will return that same error.

When multiple traces are set on *name*, then for *enter* and *enterstep* operations, the traced commands are invoked in the reverse order of how the traces were originally created; and for *leave* and *leavestep* operations, the traced commands are invoked in the original order of creation.

The behavior of execution traces is currently undefined for a command *name* imported into another namespace.

**trace add variable *name ops commandPrefix***
Arrange for *commandPrefix* to be executed whenever variable *name* is accessed in one of the ways given by the list *ops*. *Name* may refer to a normal variable, an element of an array, or to an array as a whole (i.e. *name* may be just the name of an array, with no parenthesized index). If *name* refers to a whole array, then *commandPrefix* is invoked whenever any element of the array is manipulated. If the variable does not exist, it will be created but will not be given a value, so it will be visible to **namespace which** queries, but not to **info exists** queries.
*Ops* indicates which operations are of interest, and is a list of one or more of the following items:

**array**
Invoke *commandPrefix* whenever the variable is accessed or modified via the **array** command, provided that *name* is not a scalar variable at the time that the **array** command is invoked. If *name* is a scalar variable, the access via the **array** command will not trigger the trace.

**read**
Invoke *commandPrefix* whenever the variable is read.

**write**
Invoke *commandPrefix* whenever the variable is written.

**unset**
Invoke *commandPrefix* whenever the variable is unset. Variables can be unset explicitly with the **unset** command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to *commandPrefix* so that the actual command is as follows:

```
commandPrefix name1 name2 op
```

*Name1* and *name2* give the name(s) for the variable being accessed: if the variable is a scalar then *name1* gives the variable's name and *name2* is an empty string; if the variable is an array element then *name1* gives the name of the array and name2 gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then *name1* gives the array name and *name2* is an empty string. *Name1* and *name2* are not necessarily the same as the name used in the **trace variable** command: the **upvar** command allows a procedure to reference a variable under a different name. *Op* indicates what operation is being performed on the variable, and is one of **read**, **write**, or **unset** as defined above.

*CommandPrefix* executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a Tcl procedure, then *commandPrefix* will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *commandPrefix* invokes a procedure (which it normally does) then the procedure will have to use **upvar** or **uplevel** if it wishes to access the traced variable. Note also that *name1* may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the **upvar** command.

For read and write traces, *commandPrefix* can modify the variable to affect the result of the traced operation. If *commandPrefix* modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from *commandPrefix* is ignored except that if it returns an error of any sort then the traced operation also returns an error with the same error message returned by the trace command (this mechanism can be used to implement read-only variables, for

example). For write traces, *commandPrefix* is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, *commandPrefix* will have to restore the old value of the variable.

While *commandPrefix* is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by *commandPrefix* will occur directly, without invoking *commandPrefix* (or any other traces) again. However, if *commandPrefix* unsets the variable then unset traces will be invoked.

When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked. Any errors in unset traces are ignored.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the **trace remove variable** command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

This command returns an empty string.

**trace remove *type name opList commandPrefix***
Where *type* is either **command**, **execution** or **variable**.
**trace remove command *name opList commandPrefix***
If there is a trace set on command *name* with the operations and command given by *opList* and *commandPrefix*, then the trace is removed, so that *commandPrefix* will never again be invoked. Returns an empty string. If *name* does not exist, the command will throw an error.

**trace remove execution *name opList commandPrefix***
If there is a trace set on command *name* with the operations and command given by *opList* and *commandPrefix*, then the trace is removed, so that *commandPrefix* will never again be invoked. Returns an empty string. If *name* does not exist, the command will throw an error.

**trace remove variable *name opList commandPrefix***
If there is a trace set on variable *name* with the operations and command given by *opList* and *commandPrefix*, then the trace is removed, so that *commandPrefix* will never again be invoked. Returns an empty string.

**trace info *type name***
Where *type* is either **command**, **execution** or **variable**.
**trace info command *name***
Returns a list containing one element for each trace currently set on command *name*. Each element of the list is itself a list containing two elements, which are the *opList* and *commandPrefix* associated with the trace. If *name* does not have any traces set, then the result of the command will be an empty string. If *name* does not exist, the command will throw an error.

**trace info execution *name***
Returns a list containing one element for each trace currently set on command *name*. Each element of the list is itself a list containing two elements, which are the *opList* and *commandPrefix* associated with the trace. If *name* does not have any traces set, then the result of the command will be an empty string. If *name* does not exist, the command will throw an error.

**trace info variable *name***
Returns a list containing one element for each trace currently set on variable *name*. Each element of the list is itself a list containing two elements, which are the *opList* and *commandPrefix* associated with the trace. If *name* does not exist or does not have any traces set, then the result of the command will be an empty string.

For backwards compatibility, three other subcommands are available:

**trace variable *name ops command***
This is equivalent to **trace add variable** *name ops command*.

**trace vdelete *name ops command***
This is equivalent to **trace remove variable** *name ops command*

**trace vinfo *name***
This is equivalent to **trace info variable** *name*

These subcommands are deprecated and will likely be removed in a future version of Tcl. They use an older syntax in which **array**, **read**, **write**, **unset** are replaced by **a**, **r**, **w** and **u** respectively, and the *ops* argument is not a list, but simply a string concatenation of the operations, such as **rwua**.

## EXAMPLES

Print a message whenever either of the global variables **foo** and **bar** are updated, even if they have a different local name at the time (which can be done with the **upvar** command):

```
proc tracer {varname args} {
    upvar #0 $varname var
    puts "$varname was updated to be \"$var\""
}
trace add variable foo write "tracer foo"
trace add variable bar write "tracer bar"
```

Ensure that the global variable **foobar** always contains the product of the global variables **foo** and **bar**:

```
proc doMult args {
    global foo bar foobar
    set foobar [expr {$foo * $bar}]
}
trace add variable foo write doMult
trace add variable bar write doMult
```

Print a trace of what commands are executed during the processing of a Tcl procedure:

```
proc x {} { y }
proc y {} { z }
proc z {} { puts hello }
proc report args {puts [info level 0]}
trace add execution x enterstep report
x
  → report y enterstep
    report z enterstep
    report {puts hello} enterstep
    hello
```

# trace



**trace** , a built-in Tcl routine, manages traces on variables and routines.

## See Also

**An equation solver**
Illustrates the use of trace to update dependent resources.

**Traces**
Examples and discussion.

**Tracing inappropriate variable access**

**Whole-Script Tracing, by DKF**

**An example of data objects**
AM: Using trace I implemented an idea by George Howlett, the author of BLT.

**Tiny Excel-like app in plain Tcl/Tk**
Uses traces to implement cascading recalculation of the values of spreadsheet cells.

## Synopsis

**trace** *option ?arg arg ...?*

## Documentation

**official documentation**

## Description

Once a trace is defined on a routine, the trace follows the routine even if it is renamed. If a traced routine is renamed and a new command having the original name is created, the trace remains bound to the renamed command, not the new command.

**trace add** adds a trace even if it identical to an existing trace. When there are multiple identical traces, **trace remove** removes only one of them.

Example use cases for trace:

- Communicate between parts of a GUI and the internal state of the app. (Simplified MVC, observer). In general Communicate between different parts of an app without coupling them strongly.

- Compute simple constraints for a number of variables ("if this flag is on and that one is on, then no other is allowed to be set", and some such).
- Debug - call a proc when a variable is modified (detect setting from wrong routine).
- Trace works in Itcl Itcl trace but not quite trivially.
- Clean up upon deallocation of a resource.

## Order of Processing

JCW:

```
proc tellme {id a e op} {
    puts "  $id a=$a e=$e op=$op\
        ax=[info exists ::$a] ex=[info exists ::${a}($e)]"
}

proc do args {
    puts <$args>
    uplevel 1 $args
}

trace add variable a {write unset} {tellme array}
trace add variable a(1) {write unset} {tellme element}

puts [trace info variable a]
puts [trace info variable a(1)]

do set a(0) zero
do set a(1) one
do set a(2) two
do unset a(0)
do unset a(2)
do unset a

# output is:
#
#   {wu {tellme array}}
#   {wu {tellme element}}
#   <set a(0) zero>
#     array a=a e=0 op=w ax=1 ex=1
#   <set a(1) one>
#     array a=a e=1 op=w ax=1 ex=1
#     element a=a e=1 op=w ax=1 ex=1
#   <set a(2) two>
#     array a=a e=2 op=w ax=1 ex=1
#   <unset a(0)>
#     array a=a e=0 op=u ax=1 ex=0
#   <unset a(2)>
#     array a=a e=2 op=u ax=1 ex=0
#   <unset a>
#     array a=a e= op=u ax=0 ex=0
#     element a=a e=1 op=u ax=0 ex=0
```

## Variable Trace Arguments vs Upvar

Be careful how you use the name1 and name2 parameters provided to a variable trace. It's tempting to write a single handler which tracks access to multiple variables thus:

```
proc traceHandler {name _ op} {  ;# we ignore name2 for a non-array trace
    puts "trace: variable {$name} $op"
}
unset -nocomplain x y
trace add variable x {write} traceHandler
trace add variable y {write} traceHandler
incr x
#  trace: variable {x} write
incr y
#  trace: variable {y} write
```

So far, so good, but look what happens when the variable is aliased with upvar:

```
proc loop {_var first limit script} { ;# https://wiki.tcl-lang.org/2025
    upvar 1 $_var var
    for {set var $first} {$var < $limit} {incr var} {
        uplevel 1 $script
    }
}
loop x 0 2 {
  ;# no loop body - we just want to see what happens to x
}
#  trace: variable {var} write
#  trace: variable {var} write
#  trace: variable {var} write
```

The trace fires as expected ... but the name is now coming from somebody else's code. This is sufficient if you just want to access the variable by upvar or uplevel, but useless for identifying changes to different variables managed by the same trace handler.

Bugs stemming from code not considering this behaviour can lie dormant for a very long time and be mystifying when they arise. Therefore, when the name of the variable matters to the trace handler, it's a good idea (and good practice for other kinds of callbacks) to pass that name explicitly, rather than relying on the arguments supplied by the trace:

```
proc traceHandler {name} {
    puts "trace: $name"
}
trace add variable x {write} [lambda {varname args} {traceHandler $varname} x]
```

PYK: I prefer an example that uses apply directly, because its execution namespace can be specified, and in contrast with lambda (at least as currently defined on that page), its uplevel is still the context that triggered the trace:

```
trace add variable x write [
    list apply [
        list {varname args} {traceHandler $varname} [namespace current]] x]
```

aspect: updated the lambda page to more prominently feature the Tcllib package which supports a namespace argument.

## Local Variable Traces

On comp.lang.tcl, 2004-05, CLN answers Erik Leunissen's question: Erik Leunissen wrote:

```
> The following passage from the man page to the trace command is
> mystifying to me:
>
> "If an unset occurs because of a procedure return, then the trace will
> be invoked in the variable context of the procedure being returned to:
> the stack frame of the returning procedure will no longer exist."
>
> I've read it again and again; I can't imagine how a procedure return
> affects/causes an unset.
> ...
```

```
   proc foo { } {
      set x 1
      trace add variable x unset {some code here}
   }

When foo is invoked, x is created (locally), has a trace associated with
it, then becomes unset as foo exits.
```

## Arrays

For variables that are not arrays the value of *name2* is the empty string when the trace is called, but the empty string is a perfectly valid array index (it is also a valid array variable name), so an empty value for *name2* doesn't necessarily indicate that the traced variable is scalar. To determine whether a variable is an array, use:

```
if {[array exists $varname]} {...}
```

All of the following operations result in the argument values a {} u:

```
unset a ;# regular var a
array unset a
unset a()
```

including an empty index string. array exists always returns false for the first two cases, and true for the third (even if the empty index was the only array element). There is no way for the trace to be sure which operation was performed.

Lars H: Hmm... might this be a sign that the format of these parameter lists is not well designed? An alternative would have been to put the operation first and the variable name second, so that there needn't be an index for non-array accesses. Probably too late to change now, though. (Adding a second interface which is just like the current except that it produces parameter lists in a new format is possible, but probably seen as superfluous.)

## Arrays and Upvar

A possibly surprising result that is documented for upvar:

```
If  otherVar  refers  to an element of an array, then variable traces
set for the entire array will not be invoked when myVar is accessed (but
traces on the particular element will still be invoked).
```

This means that using a trace on an array is *not* sufficient to capture all the ways elements can be accessed. For instance:

```
package require lambda
unset -nocomplain a x y
array set a {x 1 y 2}
trace add variable a {array read write unset} [lambda args {puts "::a trace:
$args"}]
```

or easier - without the need for package lambda:

```
unset -nocomplain a x y
array set a {x 1 y 2}
trace add variable a {array read write unset} {apply {args {puts "::a trace:
$args"}}}
```

Any direct use of a will trigger the trace:

```
% incr a(x)    ;# existing element
::a trace: a x read
::a trace: a x write
 2
% incr a(z)    ;# new element
::a trace: a z read
::a trace: a z write
 1
% unset a(x)
::a trace: a x unset
```

so far so good. But throw upvar into the mix:

```
% upvar 0 a(y) y    ;# existing element
% upvar 0 a(w) w    ;# new element
% incr y
 3
% incr w
 1
% unset y
% unset w
```

No traces were fired! If you're tracing an array whose elements might be upvared, beware.

## What about a "variable create" Trace?

male 2006-01-24: I wanted to trace the creation of an array element and used the write event, but ... the write event is fired after the new array element was already created! What's about a new event like "create"? Since a trace may be created on non-existent variables, this could be useful not only for arrays.

---

Donald Arseneau: Yes, write traces fire after the variable has already been set, so if you want validation of variables' values, in analogy with Tk's entry validation, then you must maintain shadow copies of the previous values, in order to undo improper settings.

## Triggering Traces when using a variable at the C level

On comp.lang.tcl, Kevin Kenny answers someone wanting to link a C variable and a Tcl variable, and have a Tcl proc invoked when the C code modified the variable:

Look in the manual for Tcl_UpdateLinkedVar. The Tcl engine has no way of telling that you've changed the variable in C; if you call Tcl_UpdateLinkedVar, that tells it your value has changed, and it fires the traces.

## Simple file I/O in traces:

```
trace add variable stdout write {puts stdout $stdout ;#}
trace add variable stdin  read {gets stdin  stdin   ;#}
```

The variables are named like the file handles. Little demo, that waits for input and prints it capitalized:

```
set stdout [string toupper $stdin]
```

## Managing Traces

Traces are like widgets and images in that they are resources that can be leaked and/or need clean-up. Counter-intuitive results sometimes arise because traces are additive rather than substitutive; a particular trace can fire a whole chain of commands. To wipe the global space clean of traces,

```
foreach variable [info glob] {
    foreach pair [trace info variable ::$variable] {
        lassign $pair op traced_command
        trace remove variable ::$variable $op $traced_command
    }
}
```

## Swallowed Errors: Command Delete Traces

PYK 2015-04-02:

Errors are ignored in command delete traces:

```
variable var1 {1 one 2 two}
proc p1 args {}
#intentional bad code here: the commandPrefix doesn't have the right command
signature.
trace add command p1 delete [list dict unset var1 1]
rename p1 {}
puts $var1 ;# -> 1 one 2 two
```

This is the current design, not a bug. Anyone have any info on the rationale?

## Traces of Command Executions

### step traces

**enterstep** and **leavestep** traces fire for all steps, recursively. When this is undesired, the depth of the recursion can be constrained by having the trace procedure look at info level.

---

Donald Arseneau: Another tricky trap is that errors in traces may give error messages, but no context; the only context is for whatever triggered the trace. Thus, if you ever see Tk error messages like

```
can't set "xv": invalid command name "oops"
    while executing
"incr xv"
```

then you should look for a variable trace on the xv variable.

---

Schnexel: Oh the tricky trace traps! I tried to automaticly update a derivedData array by setting a trace on the parentData array (scenario simplified)... Now I get a surreal result:

```
set bla "What happened:\n"

namespace eval bbb {

    array set lala [list 1 v1 2 v2 3 v3]
    trace add variable ::bbb::lala {read array} ::bbb::tra
    proc tra args {
        append ::bla "\n (TRACE $args)"
        array unset ::bbb::lala                         ;# also deletes trace (yet
the "array" op still fires)
        foreach {n v}  [list 1 trv1 2 trv2 3 trv3] { set ::bbb::lala($n) $v }
    }
}

namespace eval aaa {
    append ::bla "\n\[info exists ::bbb::lala(1)\]==..."; append ::bla ... [info
exists ::bbb::lala(1)]
    append ::bla "\n\[info exists ::bbb::lala(1)\]==..."; append ::bla ... [info
exists ::bbb::lala(1)]
    append ::bla "\n\$::bbb::lala(1)==...";              append ::bla ...
$::bbb::lala(1)
}

puts $bla
```

which gives the output

```
What happened:
[info exists ::bbb::lala(1)]==...
 (TRACE ::bbb::lala 1 read)
 (TRACE ::bbb::lala {} array)...0
[info exists ::bbb::lala(1)]==......1
$::bbb::lala(1)==......trv1
```

So, upon first read access of lala, it does not exist anymore, whilst upon second read it is there. Can anybody make sense of this?

Lars H: Regarding why the "array" op fires: It it fires at the very array unset ::bbb::lala where you comment upon this, i.e., before the foreach, which is consistent with the trace manpage (only read and write traces are disabled inside a read or write trace). But why info exists reports 0 I don't know... Perhaps some caching issue (the variable that was looked up is not the one that is there when the result is returned)? You'll probably need to read the source to find out.

Schnexel: Wrrr... Here´s a simpler example. Array trace is bugged!

```
array set ::lala [list 1 v1  2 v2]
array set ::lala [list 3 v3  4 v4]
puts "\$::lala==[array get ::lala]"                                    ;# O.K.

trace add variable ::lala read ::tra


proc tra args {
    puts "    (TRACE $args)"
    trace remove variable ::lala read ::tra

    array set ::lala   [list A trvA  B trvB]
    puts "    within trace:  \$::lala==[array get ::lala]" ;# O.K.
}

puts "1st read outside:  \$::lala==[array get ::lala]"      ;# not O.K. !
puts "2nd read outside:  \$::lala==[array get ::lala]"      ;# O.K.
```

Output:

```
$::lala==4 v4 1 v1 2 v2 3 v3
reading ::lala
    (TRACE ::lala 4 read)
    within trace:  $::lala==4 v4 A trvA 1 v1 B trvB 2 v2 3 v3
1st read outside:  $::lala==4 v4 1 v1 2 v2 3 v3
2nd read outside:  $::lala==4 v4 A trvA 1 v1 B trvB 2 v2 3 v3
```

## Tracking where procedures were defined

DKF: One of the neat things about Tcl is that you can attach traces to things that in most other languages would be completely impossible to track. Here's how to find out where procedures are defined, by using an execution trace on [**proc**] itself.

```
proc recordDefinitionLocation {call code result op} {
    if {$code} return
    set name [uplevel 1 [list namespace which -command [lindex $call 1]]]
    set location [file normalize [uplevel 1 {info script}]]
    puts stderr "defined '$name' in '$location'"
}
trace add execution proc leave recordDefinitionLocation
```

Trying it out…

```
% parray tcl_platform
defined '::parray' in
'/Library/Frameworks/Tcl.framework/Versions/8.6/Resources/Scripts/parray.tcl'
tcl_platform(byteOrder)     = littleEndian
tcl_platform(machine)       = x86_64
tcl_platform(os)            = Darwin
tcl_platform(osVersion)     = 12.5.0
tcl_platform(pathSeparator) = :
tcl_platform(platform)      = unix
tcl_platform(pointerSize)   = 8
tcl_platform(threaded)      = 1
tcl_platform(user)          = dkf
tcl_platform(wordSize)      = 8
```

That looks correct for my system.

## Proposal: Modify trace command ... enter to Act as a Command Filter

PYK 2013-12-22: The result of a command run as a trace is currently discarded. It could instead be used as the command to actually call. For example, the result of the following script would be 12, not 21

```
proc add args {
    ::tcl::mathop::+ {*}$args
}

trace add execution add enter {apply {{cmd op} {
    set args [lassign $cmd name]
    foreach arg $args[set args {}] {
        if {$arg % 2 == 0} {
            lappend args $arg
        }
    }
    return [linsert $args 0 $name]
}}}

add 1 2 3 4 5 6
```

## Strange problem I had with trace and variables in namespaces

SVP 2016-12-22:

An interesting bug I have to fix.

I have a couple of "shapescale" widgets in an application. These are pure tcl, and they act like slider controls but with a non rectangular shape. More importantly they monitor their variable with a trace, so when ever the value changes the trace proc will redraw the widget:

```
proc _variableWriteTrace {w name1 name2 op} {
    variable Priv

    debugcallinfo traces
    # Array element?
    variable $name1
    if {$name2 ne ""} {
        set name1 "${name1}($name2)"
    }
    # Check the variable exists before we go reading from it
    debugputs traces "  +->Checking if variable $name1 exists ..."
    if {[info exists $name1]} {
        debugputs traces "  +->yes; value = '[set $name1]'"
        set Priv($w,value) [set $name1]
        _redrawSlider $w
    }
}
```

In theory.

But my problem was that when the data changed, and there were a whole bunch of other variables that changed, and a whole bunch of other traces fired, and whole bunch of other widgets were updated, except for the two shapescale widgets. Well, sometimes they would update. That really was the problem.

Now after much investigation I found the following behaviour. This example works. The variableWriteTrace proc can see the variable and it's value and it redraws the slider:

```
(System32) 51 % set ::Gap12::gap12_0(config_volume) 7
[::ShapeScale|traces] _variableWriteTrace
w='.pr1.ed.pane.pafr.gap12.tree.config_volume' name1='::Gap12::gap12_0'
name2='config_volume' op='write'
[::ShapeScale|traces]{  +->Checking if variable ::Gap12::gap12_0(config_volume)
exists ...}
[::ShapeScale|traces]{  +->yes; value = '7'}
```

But then this second example fails. And this is the important one because the proc that changes the value lives in another namespace altogether called "Gap12", part of another code module:

```
(System32) 52 % namespace eval ::Gap12 { set gap12_0(config_volume) 7 }
[::ShapeScale|traces] _variableWriteTrace
w='.pr1.ed.pane.pafr.gap12.tree.config_volume' name1='gap12_0'
name2='config_volume' op='write'
[::ShapeScale|traces]{  +->Checking if variable gap12_0(config_volume) exists ...}
```

So the trace is called, but without the namespace prefix. Hence the variable being referenced doesn't exist, and so the value has changed but no redraw happens and the widget looses synch with the data.

Now the trace info tells me that the trace is there. I can also see a second unrelated trace which monitors for project data changes.

```
(System32) 53 % trace info variable ::Gap12::gap12_0(config_volume)
{write {::Gap12::_changedDataVariableTrace .pr1.ed.pane.pafr.gap12}} {write
{::ShapeScale::_variableWriteTrace .pr1.ed.pane.pafr.gap12.tree.config_volume}}
```

And this is the code that sets the trace:

```
trace add variable [set Priv($w,-variable)] write
"::ShapeScale::_variableWriteTrace $w"
```

And parray shows that the variable name is prefixed with the namespace correctly:

```
::ShapeScale::Priv(.pr1.ed.pane.pafr.gap12.tree.config_volume,-variable)
= ::Gap12::gap12_0(config_volume)
```

So the problem is that despite the fact that I use the fully qualified name of the variable in the trace add command, trace will still call the handler WITHOUT the namespace prefix when I access the variable within it's own namespace without the namespace qualification.

This seems a little strange to me?

# Traces



## Summary

How to use traces in Tcl

## Description

## Read-Only Variable

DKF: A simple way to get a read-only global variable is to use a suitable trace:

```
set ROglobal someValue
trace add variable ROglobal write "[list set ::ROglobal $ROglobal];error read-
only;#"
```

Notice the time- and hair-saving tricks:

1. Using [list] to construct a *guaranteed* safe command for later execution.
2. Using colon notation to force a reference to a global variable, whatever the context.
3. Inserting the global name of the variable in the trace command, instead of working with its local referent.
4. Using a trailing ";#" to trim the undesirable extra arguments from the trace command.

RWT: This is great code, but hard to find if you're thinking of a constant or constants or C's #define. (There! Now the Wiki search will find this page!)

## Variable aliases

JCG: I remember having had trouble when the trace was set on **::var**, yet usage was based on **global var**, or the other way around. Does anyone know what the exact behavior is when mixing these two approaches?

DKF: The thing to remember is that traces are always called in the context of the operation that is performing the read, write or unset, and the standard parameters passed in from the Tcl system refer to the way in which the variable was referred to *in that context*. Whenever you are setting up a trace on a variable that exists independently of the stack (i.e. a global or namespace variable - the two are really the same thing anyway) then it is much less hassle to pass a fixed name for that variable in as one of your own parameters to the trace (you can sort-of form closures in Tcl using appropriate scripts, and some of the more cunning lambda-evaluation schemes I've seen have been based on this.) The only time this is not going to work (in standard Tcl) is when you are dealing with variables that only exist on the call-stack - procedure locals. With these, you have to

carefully build an [upvar] reference to the variable and then use that name to figure out what is going on. This is not easy, but it has been adequately covered in many books, like BOOK Tcl and the Tk Toolkit which is where I learnt all this stuff (if I can remember that far back!)

JC: Donal, you might be talking about the following:

```
trace add variable ::var write monitor
proc monitor {name args} {
        upvar $name value
        puts "'[info level 1]' changes '$name' to '$value'"
}

### Change 1
proc change1 {} {
        set ::var bar
}
change1

### Change 2
proc change2 {} {
        global var
        set var foo
}
change2

### Change 3
proc change3 {} {
        upvar var local
        set local bar
}
change3
```

The above code produces the following output:

```
'change1' changes '::var' to 'bar'
'change2' changes 'var' to 'foo'
'change3' changes 'local' to 'bar'
```

The problem is that there is no way of knowing which variable is actually being modified, all one has access to is its alias name as used in the *set* command which triggers the trace, i.e., in *monitor* the parameter *name* is equal to *local*, how do you resolve that name back to the original *::var* variable?

DKF: There is *no* standard mechanism for doing this. So I instead pass a globally-valid (fully-quantified) variable name as part of the trace script that I supply. Like that, I don't care about how the variable was accessed since I can always access it myself using a clearly defined route. And, because of the restrictions involved with the use of variables and Tk, I find sticking to only putting traces on globally-valid names to not be a problem.

Note that [namespace current] is very useful when you want to put in code to automatically figure out what the globally-valid variable name actually is.

ulis: If I understand well 'there is no way of knowing which variable is actually being modified', I disagree. Manual said:

"When the trace triggers, three arguments are appended to command so that the actual command is as follows: **command name1 name2 op**.

*Name1 and name2 give the name(s) for the variable being accessed*: if the variable is a scalar then name1 gives the variable's name and name2 is an empty string; if the variable is an array element then name1 gives the name of the array and name2 gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then name1 gives the array name and name2 is an empty string. Name1 and name2 are not necessarily the same as the name used in the trace variable command: the upvar command allows a procedure to reference a variable under a different name. Op indicates what operation is being performed on the variable, and is one of read, write, or unset as defined above."

You know exactly under which name the variable is modified and you can use this name in combination with **upvar**.

UK Ah i think you missunderstand. The issue lies in another detail:

```
trace add variable <myarray> write mytraceproc
mytraceproc {_a e op} {
      switch $e \
        limits,max {
      } limits,min {
      } current {
      } name {
      } unit {
}
```

if you now do something like

```
upvar ::myarray(name) name
set name "Temperature in Reactor"
```

all your nice planning is broken for good

PYK 2012-12: So don't do that :)

---

DKF: If you want to set up a trace on an expression, then you implement this using the style advocated in the following example:

Suppose you want to have a trace on a pair of variables such that some callback (which I'll imaginatively describe as *callback* here) is called whenever some complex set of conditions (either *$VAR_I>0 && $VAR_B=="true"* or *$VAR_I>=10 && $VAR_B=="false"*) then you would do it like this:

```
trace add variable VAR_I write doTest
trace add variable VAR_B write doTest
proc doTest args {
    upvar #0 VAR_I i VAR_B b
    if {
        ($i>0 && [string equal $b "true"]) ||
        ($i>=10 && [string equal $b "false"])
    } then {
        upvar #0 callback
    }
}
```

You can't extend this to a general watch of an expression, since an expression can include a call to a general command, and solving exactly what variables a general command accesses is tough. Even if we restricted ourselves to procedures, we would still need to solve the Halting Problem (a famously insoluble thing in Computer Science, where there is a *proof* of insolubility...) Since you can usually tell straight off what variables in an expression actually matter, you can shortcut all that stuff and just tell Tcl exactly what to watch yourself...

---

DKF: Come on people, this page isn't just "Ask Dr. Donal" - fill in more of this stuff yourselves!

---

KCH: Here is an idea I have been batting around for using traces to create a kind of super-set of events. Yes, Tcl gives you the ability to define virtual events, but they must be defined in terms of existing X-type events. I was looking for the ability to fire off an event that is completely unrelated to any existing Tcl events, that says, for example, "you have a new meeting scheduled and the details are attached". Something like the following seems like it will probably work, but my gut tells me there is much room for refinement.

The basic idea is to use a namespace to wrap the "Uber-events", use a set of namespace variables with write traces attached to track the events and the "details" attached to an event, and namespace procs to register for, trigger, or delete the events.

```
namespace eval ::UberEvents {
    namespace export RegisterForEvent \
                     DeleteEvent      \
                     GetEventDetails  \
                     TriggerEvent

    set Events(List) {}

    ##########################################################################
    #
    # RegisterForEvent
    #
    # Registers for the UberEvent named $EventName, so that Command is
    # called when the event is "triggered."
    #
    ##########################################################################
    proc RegisterForEvent {EventName Command} {
        variable ::UberEvents::Events
        set ID [trace add variable ::UberEvents::$EventName {write} "[list eval
$Command];#"]
        lappend ::UberEvents::Events(List) [list $EventName $ID $Command]
        return $ID
    }

    ##########################################################################
    #
    # DeleteEvent
    #
    ##########################################################################
    proc DeleteEvent {EventName} {
        unset ::UberEvents::$EventName
    }

    ##########################################################################
    #
    # GetEventDetails
    #
    # Fetches the details of the event.
    #
    ##########################################################################
    proc GetEventDetails {EventName} {
        upvar #0 ::UberEvents::$EventName Details
        return $Details
    }

    ##########################################################################
    #
    # TriggerEvent
    #
    # Triggers the UberEvent named $EventName, giving it $Data
    #
    ##########################################################################
    proc TriggerEvent {EventName {Data ""}} {
        set ::UberEvents::$EventName $Data
    }
}
```

Then you can import the namespace and use

```
RegisterForEvent NewMeeting MyMeetingCallback
```

to set up to be informed whenever the NewMeeting event triggers, and

```
TriggerEvent NewMeeting "Details Of The Meeting"
```

will trigger the NewMeeting event, and everyone registered for that event will be notified. GetEventDetails will return the data associated with the event.

```
DeleteEvent
```

is used to remove the event and all associated callbacks.

This can easily be extended to have a common "event handler" which is set up to be called periodically, triggering the event when the handler detects it has occurred. The event details could be passed automatically to each of the callbacks, similar to the way that Tcl handles events, but then you would probably have cases where you would end up putting ";#" at the end of your callback the same way as was demonstrated in code at the beginning of this page. Making the extra data optional but retrievable is *imho* preferable....

Feedback?

jmn: While using trace here seems fine, I'm not sure exactly what it buys you. You have the command stored in a list, why not just uplevel #0 it from within TriggerEvent? (or use 'after 0' if TriggerEvent needs to return first...(?))

KCH: I want to support multiple "sinks" for an event, just as you can have multiple traces running on a variable simultaneously. I created a version which uses the following for TriggerEvent. The only differences between V1 & V2 is V2 doesn't create the traces, and V2 uses a foreach loop to find and execute each matching registered callback.

```
proc TriggerEvent {EventName {Data ""}} {
    set ::UberEvents::$EventName $Data

    foreach Event $::UberEvents::Events(List) {
        if {$EventName eq [lindex $Event 0]} {
            eval [lindex $Event 2]
            }
        }
    }
```

Note that I keep the set of the variable so that the commands can still access the data passed to TriggerEvent. The timing (Tcl/Tk 8.4.6.0, ActiveState build, WinXP, 2.8GHz, 1G Ram) shows the trace implementation to be faster, though I haven't done enough to figure out if the relationship is linear or something other. I suspect that there may be something faster than "eval", but for now the simplicity makes trace more atractive to me. The timing I got follows:

```
UberEvents V1
-------------
1   628
2  1263
3  1871
4  2494
5  3124
6  3757
7  4420
8  5009
9  5622
10 6149

UberEvents V2
-------------
1   647
2  1268
3  1994
4  2524
5  3288
6  3832
7  4525
8  5371
9  5779
10 6269
```

I might be able to get a speedup using a nested foreach to extract the sublist elements, but....

---

Ken: Just wondering, trace allows us to monitor a variable and if write, can help us to execute commands. But the one I did can only execute once. Is there any reason why is it so? If no, how must I code it to execute multiple times wheever the variable is been written?

Donald Arseneau It should work each time. There must be a problem with your specific code. See my next addition for an example, though not using exec.

---

Donald Arseneau Here is a toy example for synchronizing variables with external data. This method is most useful for interacting with a database server, or with instrumentation, but for this sample let's synchronize with file modification times

```
proc trace_mtime { var file op } {
    upvar $var v
    switch $op {
        read {
            # read the variable, so synchronize to the file's mtime
            set v($file) [file mtime $file]
        }
        write {
            # Set the variable means to set the file's mtime
            if { ![file exists $file] } {
                # No file, create an empty one
                close [open $file w]
            }
            # Set its modification time
            file mtime $file $v($file)
        }
    }
}

trace add variable mtime {read write} trace_mtime
```

Report actual file modification time:

```
puts "File foo.bar was modified at [clock format $mtime(foo.bar)]"
```

Pretend file log.log was modified yesterday:

```
set mtime(log.log) [clock scan yesterday]
```

Note that there is very little error checking. Reading the mtime for a non-existant file, or setting an invalid mtime value result in reasonable error messages:

```
can't set "mtime(log.log)": expected integer but got "foobar"
can't read "mtime(dddd)": could not read "dddd": no such file or directory
```

cstephan A simple example of how to utilize trace to provide event processing

```
# some implementations with 'interp -safe' utilize tcl_trace inside the interp.
set traceCommand [info command *trace]

namespace eval eventTest {

   global traceCommand
   variable processComplete {}

   namespace export whenDone

   proc whenDone {procName} {

       variable processComplete
           global traceCommand

           set callingNamespace [uplevel {namespace current}]

       $traceCommand variable processComplete w \
             [join [list $callingNamespace :: $procName] {}]

   }

   proc finally {name1 name2 op} {

     puts "proc finally() executed"
          if {[catch {puts "op:$op -- $name1($name2): [subst $[list $name1]
($name2)]"}]} {
                # IS SCALAR:
                puts "op:$op - $name1:[subst $[list $name1]]"
          }
          puts {}
   }

   $traceCommand variable ::eventTest::processComplete w ::eventTest::finally

}

proc doFirst {name1 name2 op} {
  puts "proc doFirst() executed"
  # TRY RETURNING INDEX NAME2 OF ARRAY NAME1
  if {[catch {puts "op:$op -- $name1($name2): [subst $[list $name1]($name2)]"}]} {
    # NAME1 IS NOT AN ARRAY, OUTPUT NAME1 AS SCALAR
    puts "op:$op - $name1:[subst $[list $name1]]"
  }
  puts {}
}
proc doSecond {name1 name2 op} {
  puts "proc doSecond() executed"
  # TRY RETURNING INDEX NAME2 OF ARRAY NAME1
  if {[catch {puts "op:$op -- $name1($name2): [subst $[list $name1]($name2)]"}]} {
    # NAME1 IS NOT AN ARRAY, OUTPUT NAME1 AS SCALAR
    puts "op:$op - $name1:[subst $[list $name1]]"
  }
  puts {}
}
proc doThird {name1 name2 op} {
```

```
  puts "proc doThird() executed"
  # TRY RETURNING INDEX NAME2 OF ARRAY NAME1
  if {[catch {puts "op:$op -- $name1($name2): [subst $[list $name1]($name2)]"}]} {
    # NAME1 IS NOT AN ARRAY, OUTPUT NAME1 AS SCALAR
    puts "op:$op - $name1:[subst $[list $name1]]"
  }
  puts {}
}

# register test events
#(enclosed this in a proc because from tclsh they halt paste)
proc regEvents {} {
  ::eventTest::whenDone ::doFirst
  ::eventTest::whenDone ::doSecond
  ::eventTest::whenDone ::doThird
}
regEvents
```

Watch procedures get notified of the transaction against the processComplete variable:

```
$ set ::eventTest::processComplete {true}

proc doThird() executed
op:w - ::eventTest::processComplete:true

proc doSecond() executed
op:w - ::eventTest::processComplete:true

proc doFirst() executed
op:w - ::eventTest::processComplete:true

proc finally() executed
op:w - ::eventTest::processComplete:true

true
```

---

glennj See also http://www.rosettacode.org/wiki/Defining_Primitive_Data_Types#Tcl for a demonstration of creating a "primitive data type" using traces.

---